ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 10, No. 2, 2025

A COMPARATIVE STUDY OF INFERENCE FRAMEWORKS FOR NODE.JS MICROSERVICES ON EDGE DEVICES

Oleh Chaplia[1], Halyna Klym[1], Kateryna Babii[2]

[1]Lviv Polytechnic National University, 12, S. Bandery str., Lviv, 79013, Ukraine,
[2]Eleks Inc., 10091 Park Run Drive, Suite 200, Las Vegas, NV 89145, USA.
Authors' e-mails: *oleh.y.chaplia@lpnu.ua, halyna.i.klym@lpnu.ua, kateryna.babii@eleks.com*

https://doi.org/10.23939/acps2025.02.233

Submitted on 11.09.2025

***Abstract*: Deploying small language models (e.g., SLMs) on edge devices has become increasingly viable due to advancements in model compression and efficient inference frameworks. Running small models offers significant benefits, including privacy through on-device processing, reduced latency, and increased autonomy. This paper conducts a comparative review and analysis of Node.js inference frameworks that operate on-device. It evaluates frameworks in terms of performance, memory consumption, isolation, and deployability. The paper concludes with a discussion and decision matrix to guide developers toward optimal choices. This approach pushes microservices one step closer to becoming first-class intelligent services rather than clients of external AI.**

***Index terms*: microservices, small language models, edge computing, artificial intelligence, benchmarking, distributed systems.**

I. INTRODUCTION

The rise of large language models has sparked interest in integrating AI capabilities into microservice architectures [1]. Traditionally, incorporating LLMs into a system meant treating the model as a separate service. For example, a dedicated REST API or sidecar that the main application calls for inference. This service-oriented approach decouples AI processing, but it introduces network or inter-process communication overhead, added complexity in deployment, and potential latency bottlenecks [2]. As applications move toward the edge, these overheads and complexities become more pronounced due to limited hardware resources and the need for offline, low-latency operation [3].

Deploying small language models (SLMs) on resource-constrained edge hardware is becoming increasingly feasible as model efficiency improves [4]. Running SLM inference on devices like the Raspberry Pi 5 offers privacy (as no data leaves the device) and low latency (since there is no cloud round-trip) [5]. In a secure microservice architecture, each service should run in isolation, preventing memory or execution interference.

However, this scenario also poses unique challenges. The frameworks used for local inference must be lightweight, efficient, and secure [6]. Security demands memory isolation, preventing a compromised model process from affecting the host service. Equally important is the ability to deploy and integrate easily with microservices, whether via direct bindings, HTTP APIs, or Docker containers [7]. Microservices can be developed in various programming languages, including JavaScript, Python, Go, Java, and Rust, depending on the system's requirements and ecosystem. One of the possible tools, Node.js, offers distinct advantages for microservice development in the scope of asynchronous, event-driven architecture, which supports high concurrency, lightweight deployments, and integration with modern AI and inference libraries [8].

Therefore, this research explores the paradigm of embedding the inference engine directly inside the Node.js microservice as a co-located component. By integrating the model runtime in-process, we essentially create a self-contained microservice, where AI model inference is a native component of the service's logic rather than an external dependency. The result is a microservice that is AI-driven, similar to an AI agentic approach, with lower latency, fewer moving parts, and a clearer security posture.

To experiment with this global idea, this research provides a comparison of frameworks for local LLM inference that are suitable for Node.js microservices on edge hardware. This research is intentionally focused on small language models and on-device inference, without relying on external API calls.

A list of frameworks includes node-llama-cpp, Transformers.js, ONNX Runtime, and WasmEdge (WASI-NN). Within the scope of research, the design, performance on CPUs, technical qualities, security features, and deployment options of each framework are evaluated. As a result, we present a comparison table of key metrics and insights extracted from the testing and evaluation. A decision matrix for selecting the right solution is based on the key insights acquired. This research lays the foundation for further exploration of creating more intelligent small models and applying reasoning models to solve various microservice tasks.

II. LITERATURE REVIEW AND PROBLEM STATEMENT

Current efforts are focused mainly on LLMs, which are served and inferenced by specialized, separate, highly efficient inference frameworks like vLLM. In contrast,

small language models within the same system are often served by separate tools, such as Ollama or LocalAI [9].

Both approaches divide microservices and the model inference into separate components. These tools present an OpenAI-like API for local models but run the heavy inference in an external process (or even container), meaning the Node.js microservice must communicate over HTTP to get AI responses. This approach is suitable for most existing cloud-native distributed systems [10].

While this separation can simplify design in cloud settings, it is suboptimal for specialized and edge deployments. The overhead of serializing requests and responses can significantly impact latency and throughput on a low-power device. Additionally, maintaining a separate AI service complicates deployment for edge nodes, which benefit from simplicity and autonomy.

Model quantization plays a central role in achieving efficiency. Techniques such as 8-bit, 5-bit, or 4-bit quantization reduce model size and memory bandwidth with minimal loss in quality. Llama.cpp, the pioneering C++ engine for running Meta's LLaMA models, popularized 4-bit quantization via the GGML and GGUF model formats. In addition to quantization, multi-threading, and CPU SIMD optimizations, this allows even ARM-based processors to perform usable inference at rates of several tokens per second [11].

When we look a little further into the future, the evolution extends beyond just using models to also include actionable AI agents and agentic systems. These systems can act and reason by themselves. As systems become increasingly complex, the reasoning they employ also becomes more sophisticated [12].

Recent advances have enabled the direct execution of LLMs on edge hardware. Lightweight runtimes such as llama.cpp demonstrates that large models can operate efficiently on CPUs through quantization and optimization, achieving faster inference with less memory than Python-based servers [13]. Benchmarks show that 3–4B models can generate text at several tokens per second or even faster on a Raspberry Pi 5, demonstrating that real-time inference is feasible on small devices. These breakthroughs motivate embedding LLM inference inside microservices, eliminating network overhead while maintaining privacy and operational simplicity.

Node-llama-cpp is a Node.js binding for the llama.cpp library, allowing models in GGUF/GGML format to run natively within a Node process. It supports multi-threaded CPU execution, multiple quantization levels, and even GPU acceleration through CUDA, Metal, or Vulkan backends. Its primary strengths are speed, simplicity, and tight integration with Node.js applications, exposing asynchronous APIs for prompt handling and streaming outputs. However, as a native C++ addon, it runs without sandboxing, meaning a crash or exploit in the model layer can impact the entire process. Node-llama-cpp excels in offline chatbots, IoT assistants, and local reasoning components, where performance and portability outweigh security concerns [14].

ONNX Runtime provides a general-purpose inference engine that can run models exported from frameworks such as PyTorch or TensorFlow. It supports graph optimizations, INT8/FP16 quantization, and hardware acceleration (CUDA, TensorRT, DirectML, CoreML). While ONNX models are more memory-intensive. They enable architecture-agnostic deployment and can run a wide variety of model types, including GPT, T5, and distilled variants of these models. The main advantages of ONNX Runtime are model versatility, robust optimization, and enterprise maturity. Its limitations include larger memory requirements and more complex model conversion pipelines. Typical use cases involve custom-trained models integrated into microservices or hybrid AI pipelines combining multiple models [15].

Transformers.js is Hugging Face's JavaScript runtime for running modern transformer models directly in Node.js or the browser, without Python or external servers. It supports a wide range of tasks, including text generation, embeddings, classification, image processing, audio, and multimodal models, using backends such as WebGPU, ONNX Runtime, and optimized WebAssembly to strike a balance between speed and portability. The library loads pretrained models from the Hugging Face Hub, offers a high-level, Python-like API, and provides streaming generation, tokenization, and hardware-accelerated inference where available. Its strength is ecosystem breadth and ease of use, making it ideal for developers who want cross-platform AI workloads that run natively in JavaScript environments [16].

WasmEdge (WASI-NN) represents a newer approach, executing models within a WebAssembly sandbox for enhanced isolation and security. Developers can compile LLM inference code to WASM or use precompiled GGML-based modules, executing them safely within Node.js through WasmEdge bindings. Although slightly slower than native execution, WasmEdge provides strong memory isolation, portability across architectures, and a lightweight container alternative for secure, multi-tenant edge environments. It is ideal for applications requiring data privacy, sandboxed AI logic, or deployment in untrusted environments [17].

Together, these frameworks demonstrate that inference Node.js microservices is possible. The trade-offs lie in balancing performance, isolation, model flexibility, and software architecture.

In summary, these frameworks enable edge microservices to utilize LLMs, where inference runs within the service, rather than as a separate, networked service. Embedding delivers lower latency by replacing HTTP/IPC with direct calls, boosts throughput by removing an entire service layer and enabling token streaming, simplifies deployment by bundling the runtime with the microservice (each instance carrying its own AI capacity), and strengthens data security and privacy.

III. SCOPE OF WORK AND OBJECTIVES

This study explores embedded LLM inference in Node.js microservices running on edge hardware (Raspberry Pi 5, 8 GB RAM). It focuses on a small text

model inferred entirely on-device using several frameworks that enable in-process inference. Node-llama-cpp, ONNX Runtime for Node.js, Transformers.js, and WASM runtime WasmEdge (WASI-NN) are explored. External inference servers (e.g., Ollama, LocalAI, vLLM, LMStudio, Transformer Lab) and cloud APIs (e.g., OpenAI, Azure AI Foundry, and others) are excluded to isolate the design and security aspects of embedding inference directly within the microservice process.

The study evaluates Granite-4.0-350M and Google/gemma-3-270m. A dedicated benchmark was prepared that automatically downloads the model in GGUF and ONNX formats, executes standardized tests across selected frameworks, and gathers metrics. A context window of 1024 tokens and a maximum output length of 1000 tokens were used. The evaluation prompt used for all tests was *"Write a huge paragraph in detail that fully describes generative AI models."*

Measurements include startup time, time to first token (TTFT), token generation rate (tokens/s), total inference time, per-token latency (ms/token), memory usage (peak RSS), and CPU load. Using these metrics, the comparative study assesses embeddability, performance, isolation properties, resource efficiency, quantization support, and Node.js integration. The security check examines each framework's isolation model, contrasting native bindings with sandboxing for fault containment. Finally, the research presents a performance evaluation pipeline and summarizes the findings through a comparison table and a decision matrix.

The hypothesis is that embedded inference provides microservice autonomy, establishing it as the preferred strategy for intelligent, self-contained edge microservices.

IV. EXPERIMENTAL SETUP

All experiments were conducted on a Raspberry Pi 5 equipped with a quad-core Cortex-A76 CPU (2.4 GHz) and 8 GB of RAM, as well as the onboard VideoCore VII GPU, running the Raspberry Pi OS (Bookworm), a Debian-based system optimized for the hardware. The software environment included Node.js version 24 and Python version 3.13.

A custom Node.js application with additional scripts was developed to automate the testing workflow. It downloads the required model and quantization files, executes custom tests, gathers metrics, and stores the results and logs for further analysis. During each run, the system metrics are measured.

Each test was repeated multiple times to account for cold-start and steady-state conditions. The collected data provided the basis for comparing performance, resource efficiency, and security isolation across the inference framework runtimes.

V. TESTING RESULTS

The comparative study demonstrated that the Granite-4.0-350M and Google/gemma-3-270m models deliver practical inference performance when run on a Raspberry Pi 5 device. Across the optimized execution paths tested in the test environment, the observed throughput ranged from 4 to 8 tokens per second (TPS) for these models. The highest raw speed in the evaluation was achieved by the native C++ addon utilizing the Granite-4.0-350M variant, which reached approximately ~8 TPS. This performance was attributed to aggressive GGUF quantization and optimized SIMD with multithreading kernels. Other configurations, such as the ONNX Runtime approach, averaged ~6 TPS, while the WebAssembly-sandboxed variant achieved about ~5 TPS.

At these speeds, generating a standard ~100-token response on this hardware requires between 8 and 20 seconds under typical prompt lengths. Sometimes the result may take longer, up to 40 seconds, depending on the input and output. This success aligns with vendor claims for the Granite-4.0 family, which tout "2x faster inference speeds" and "significantly reduced memory requirements" for nano-sized models compared to prior generations. The study concludes that both the Granite-4.0-350M and Google/gemma-3-270m models deliver smooth, lightweight inference when embedded directly in a Node.js microservice.

Resource efficiency on the Raspberry Pi platform was notably favorable across all implementations tested. Memory usage remained well within the platform's 8 GB RAM allowance. Peak memory consumption (RAM) varied depending on the execution path: the native addon, utilizing the Granite variant, recorded the lowest use at ~1.6 GB; the sandboxed version consumed ~1.8 GB (including sandboxing overhead); and the ONNX Runtime required the most at ~2.1 GB. Startup times (model load) were equally efficient: the native binding (Granite variant) initialized fastest in ~2.3 seconds, the sandboxed module took ~3.0 seconds, and ONNX Runtime required ~3.5 seconds due to extra time needed for session building. After initialization, all frameworks maintained stable throughput and consistent per-token latency across repeated runs, providing practitioners with actionable data for microservice-embedded local inference.

Together, these findings highlight that even modest edge hardware can host an embedded reasoning engine, enabling self-contained, intelligent microservices without reliance on external AI infrastructure.

VI. ARCHITECTURAL OVERVIEW OF THE INFERENCE FRAMEWORKS

Node-llama-cpp embeds the llama.cpp inference engine directly into Node.js as a native C++ addon, enabling extremely fast CPU inference with aggressive quantization. Its architecture is minimal, including a memory-mapped GGUF weight loader, multi-threaded SIMD kernels, and optional GPU backends. This tightly coupled native integration maximizes performance but provides no isolation, because native code executes in the same memory space as the Node.js event loop. Any segmentation fault or vulnerability in the inference engine

can cause the microservice process to crash. Architecturally, node-llama-cpp has no runtime graph abstraction, no intermediate representation, and no graph optimizations. Instead, it executes a fused attention kernel tailored for transformer architectures and optimized for quantized integer weights.

ONNX Runtime for Node.js uses a very different architecture. It loads ONNX computational graphs exported from PyTorch, TensorFlow, or other frameworks and executes them using an optimized C++ runtime that provides graph fusion, operator-level parallelism, and multiple execution providers (CPU, CUDA, TensorRT, ROCm, DirectML, OpenVINO, depending on the platform). ONNX models are fully declarative. The runtime interprets an IR graph with standardized ops, allowing a wide variety of model families beyond LLaMA derivatives. This introduces greater flexibility but also higher memory overhead, since ONNX models often contain verbose metadata, unoptimized tensors, and larger operator kernels. ONNX Runtime for Node.js is provided as a native addon, so inference executes in the same process without isolation. However, its enterprise-oriented architecture emphasizes stability, operator correctness, and predictable performance across hardware types.

Transformers.js follows a pure-JavaScript architecture defined around pluggable inference backends (WebGPU, ONNX Runtime Web, WebAssembly, and CPU-optimized WASM kernels). Unlike node-llama-cpp or ONNX Runtime, Transformers.js does not run native code directly inside Node.js. Instead, it interposes a typed model-execution layer built around the Hugging Face model hub, handling model loading, tokenization, streaming, and execution through standardized pipelines. This abstract backend architecture allows portability across browsers, desktop JavaScript, and Node environments. However, transformers.js supports a narrower range of model architectures and often relies on WebGPU or WASM acceleration for speed; without them, throughput is lower. Its architecture favors portability and developer experience over raw performance or hardware-specific optimizations.

WasmEdge with WASI-NN employs a fundamentally different design: inference is executed inside a WebAssembly virtual machine. A WASM module loads the model using plugins such as ggml-NN, TFLite-NN, ONNX-NN, or OpenVINO-NN. WASI-NN standardizes the interface between the WASM sandbox and native inference backends. The runtime provides strong sandbox isolation. Therefore, memory, syscalls, and host interaction are restricted, making it ideal for secure multi-tenant or untrusted environments. The architecture introduces overhead due to boundary crossings and data copying into WASM linear memory, but modern AOT compilation and optimized native plugins reduce this overhead. WasmEdge prioritizes security and isolation, providing process-like separation while still running inference “in-process” from the microservice’s perspective.

VII. QUALITATIVE EVALUATION AND DECISION MATRIX

This section presents a decision matrix comparing the frameworks on various qualitative aspects important for edge microservices.

Table 1

Technical comparison

Criteria	Node-llama-cpp	ONNX Runtime	WasmEdge (WASI-NN)	Transformers.js
In-process Execution	Native C++ addon	Native C++ addon	WASM runtime in-process	Pure JS with pluggable backends
Isolation	None (shares Node memory)	None (native code)	Strong WASM sandbox	JS-level safety, backend-dependent isolation
CPU performance	Very fast (SIMD, multi-thread, quantized)	Fast (graph fusion, INT8/FP16)	Good, some WASM overhead	Lower, depends on WASM, ONNX, WebGPU
GPU support	CUDA, Metal, Vulkan.	CUDA, TensorRT, DirectML	Limited, experimental	WebGPU, ONNX Runtime Web GPU
Model format	GGUF, GGML	ONNX	GGML, TFLite, ONNX, OpenVINO via plugins	HF models auto-converted to JS, WASM, WebGPU formats
Memory use	Low (mmap)	Moderate (verbose models)	Medium, High (WASM overhead)	Moderate (JS, backend buffers)

Table 2

Integration and deployment comparison

Criteria	Node-llama-cpp	ONNX Runtime	WasmEdge (WASI-NN)	Transformers.js
Installation	NPM package	NPM package	NPM package, WASM runtime, WASI-NN plugin	NPM package
Maturity	Active, open-source	Enterprise-grade (Microsoft)	Growing (CNCF)	Mature HuggingFace ecosystem
Edge suitability	Strong (quantized inference)	Good (ARM64 support)	Excellent (sandboxed)	Good (WebGPU/WASM)
Use cases	Offline IoT assistants, chatbots, speech, text, reasoning, automation	AI-driven microservices, custom models, multimodal tasks, enterprise inference services	Secure isolated edge AI, untrusted workloads, WASM-driven microservices	Browser, Node JS AI apps, web agents

Table 1 compares the technical and architectural traits of the inference frameworks. It shows that all

support true in-process execution but differ in isolation strength and hardware integration. Node-llama-cpp offers the highest CPU performance and simplest native binding, ONNX Runtime provides broad model compatibility and hardware acceleration, while WasmEdge prioritizes security with WebAssembly sandboxing at a small performance cost.

Table 2 focuses on integration and deployment factors. Node-llama-cpp is lightweight, easy to install, and ideal for quick edge deployments. ONNX Runtime is more resource-intensive but enterprise-ready and well-documented. WasmEdge, though less mature, excels at secure, portable microservices, trading simplicity for isolation.

Together, the tables highlight the trade-off between performance, flexibility, and security when embedding LLM inference into Node.js microservices.

VIII. DISCUSSION

The results confirm that embedding LLM inference directly into Node.js microservices on edge devices is not only feasible but also often offers benefits. By integrating the model into the same process, microservices become self-contained, intelligent units capable of local reasoning without network overhead or data exposure. This architectural shift simplifies deployment, reduces latency, and allows AI outputs to be tightly coupled with the service's logic. For example, enabling real-time token streaming or direct function-level interaction with the model. Although this approach blurs traditional boundaries between business logic and AI, modern frameworks like node-llama-cpp mitigate such concerns with structured output controls and schema enforcement.

The study highlights a clear trade-off between performance and isolation. Frameworks such as node-llama-cpp and onnxruntime-node provide near-native speed and efficient quantized execution, but run unsandboxed native code, requiring complete trust in the model library. In contrast, WasmEdge introduces a secure sandbox that isolates model execution from the host process, preventing potential vulnerabilities from propagating but adding a small computational overhead. This trade-off suggests that native bindings are ideal for trusted environments that prioritize speed, while WebAssembly-based inference is best suited for untrusted edge deployments that require isolation.

Finally, hardware and developer experience considerations shape the choice of framework. The Raspberry Pi 5 proved capable of handling small models with quantization. It is adequate for lightweight reasoning tasks. However, memory constraints limit the size of models, making small, efficient architectures more practical for edge AI. Node-llama-cpp offers the fastest setup and simplest integration for on-device text generation. ONNX Runtime provides broader model support and production-grade optimization for diverse workloads. WasmEdge delivers portable, secure inference, ideal for scaling across heterogeneous and potentially untrusted edge environments. Together, these findings demonstrate that embedding inference transforms conventional microservices into autonomous, intelligent agents, thereby balancing performance, flexibility, and security across various deployment scenarios.

IX. CONCLUSION

This research article presented an exploration of embedding LLM inference engines directly into Node.js microservices, thereby reframing them as intelligent microservices capable of on-board AI reasoning. Focusing on node-llama-cpp, onnxruntime-node, and WasmEdge, we demonstrated that each enables model execution with tangible benefits in latency, throughput, and data security. The exact choice of technology depended on priorities, but the fact that we have multiple viable options is a strong sign of the maturity of this approach as of 2025. As hardware improves (the next generations of edge devices will only get faster, possibly including NPUs for AI) and as models become more efficient, the case for embedded inference would grow even stronger.

Our comparison tables and decision matrix can serve as a guide for practitioners. Moving forward, several areas require further research and development. For example, tooling to monitor and manage resource usage of in-process models, techniques for seamless model upgrades in running microservices, and the exploration of hybrid models. The journey to truly intelligent microservices is just beginning, and embedding LLM inference is a pivotal step in that direction – one that we have shown is practical and advantageous today.

X. CONFLICTS OF INTEREST

The authors declare no conflicts of interest.

XI. DECLARATION ON GENERATIVE AI

During the preparation of this work, the author(s) used Grammarly to check grammar and spelling, paraphrase, and reword. After using this tool/service, the author reviewed and edited all content as needed and takes full responsibility for the publication's content.

References

[1] Patil, R. & V. Gudivada. (2024). A Review of Current Trends, Techniques, and Challenges in Large Language Models (LLMs). *Applied Sciences*, 14(5), 2074. DOI: 10.3390/app14052074.

[2] Blinowski, G., A. Ojdowska, & A. Przybylek. (2022). Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*, 10, 20357–20374. DOI: 10.1109/ACCESS.2022.3152803.

[3] Piccialli, F., D. Chiaro, P. Qi, V. Bellandi, & E. Damiani. (2025). Federated and edge learning for large language models. *Information Fusion*, 117, 102840. DOI: 10.1016/j.inffus.2024.102840.

[4] Bucher, M. J. J. & M. Martini. (2024). Fine-Tuned "Small" LLMs (Still) Significantly Outperform Zero-Shot Generative AI Models in Text Classification. DOI: 10.48550/arXiv.2406.08660.

[5] Shoop, E., S. J. Matthews, R. Brown, & J. C. Adams. (2025). Hands-on parallel & distributed computing with Raspberry Pi devices and clusters. *Journal of Parallel and Distributed Computing*, 196, 104996. DOI: 10.1016/j.jpdc.2024.104996.

[6] Alizadeh, K., I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. D. Mundo, M. Rastegari, & M. Farajtabar. (2024). LLM in a flash: Efficient Large Language Model Inference with Limited Memory. DOI: 10.48550/arXiv.2312.11514.

[7] Chaplia, O., H. Klym, & E. Elsts. (2024). Serverless AI Agents in the Cloud. *Advances in Cyber-Physical Systems*, 9(2), 115–120. DOI: https://doi.org/10.23939/acps2024.02.115.

[8] Chaplia, O., H. Klym, M. Konuhova, & A. I. Popov. (2025). Enhancing REST API Handlers Organization for Node.js Microservices. *SN Computer Science*, 6(7), 776. DOI: 10.1007/s42979-025-04311-8.

[9] Vake, D., J. Vičič, & A. Tošić. (2025). Hive: A secure, scalable framework for distributed Ollama inference. *SoftwareX*, 30, 102183. DOI:10.1016/ j.softx.2025.102183.

[10] Chen, F., L. Zhang, & X. Lian. (2021). A systematic gray literature review: The technologies and concerns of microservice application programming interfaces. *Software: Practice and Experience*, 51(7), 1483–1508. DOI: 10.1002/spe.2967.

[11] Dettmers, T., A. Pagnoni, A. Holtzman, & L. Zettlemoyer. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. DOI: 10.48550/ arXiv.2305.14314.

[12] Abdelfattah, A. S. & T. Cerny. (2023). Roadmap to Reasoning in Microservice Systems: A Rapid Review. *Applied Sciences*, 13(3), 1838. DOI: 10.3390/app13031838.

[13] López Espejel, J., M. S. Yahaya Alassan, M. Bouhandi, W. Dahhane, & E. H. Ettifouri. (2025). Low-cost language models: Survey and performance evaluation on Python code generation. *Engineering Applications of Artificial Intelligence*, 140, 109490. DOI: 10.1016/j.engappai.2024.109490.

[14] node-llama-cpp. Run AI models locally on your machine. Available at: https://node-llama-cpp.withcat.ai/.

[15] ONNX Runtime. Available at: https:// onnxruntime.ai/.

[16] Transformers.js. Available at: https://huggingface.co/docs/transformers.js/en/index.

[17] WasmEdge. Available at: https://wasmedge.org/.

Oleh Chaplia is a PhD student in the Department of Specialized Computer Systems at Lviv Polytechnic National University, where he earned his BS and MSc degrees in Computer Engineering. Since earning his master's degree in 2015, he has been working in the field of software engineering. He has extensive commercial experience in technical leadership, cloud and software architecture, research, and development of modern software systems, including those with artificial intelligence. His academic research focuses on emerging technologies, cloud computing, cloud-native systems, microservice architecture, software systems, and artificial intelligence. Oleh's experience also includes authoring research papers and technical articles, participating in international conferences, speaking at industry events, lecturing, and mentoring.

Halyna Klym – doctor of technical sciences, professor, professor of the department of specialized computer systems of the Institute of Computer Technologies, Automation and Metrology of Lviv Polytechnic National University. In 2016, she received a Doctor of Science degree in Physical and Mathematical Sciences at Lviv Polytechnic National University. She conducts lecture courses on the design of ultra-large integrated circuits and methods and means of automated design of computer systems. She is an author of more than 170 scientific articles in international publications.

Kateryna Babii received her master's degree in Computer Science from Lviv Polytechnic National University, Ukraine. She is a Senior Software Engineer at ELEKS, USA, with more than 14 years of professional experience in software engineering. Her expertise spans large-scale web systems, microfrontend and microservice architectures, and distributed applications. Her research interests include cloud computing, software performance optimization, and the integration of artificial intelligence into modern software systems. Ms. Babii has authored several technical and scientific articles and regularly contributes to international conferences and professional technology communities. She is also an active mentor, supporting programs that empower women in technology and promote engineering education.